World Models RL Tutorial

Reinforcement learning (RL) is a machine learning approach that integrates representations of past and present states and actions with strong predictive models---allowing RL agents to predict the future based on their past and present experiences[1]. RL algorithms use this artificial foresight to make decisions that maximize their rewards over time. Significantly, RL algorithms often perform well in situations where optimizing for immediate rewards has negative long-term consequences. Model-free RL algorithms, like Q-learning, tend to operate on a simple trial-and-error framework, but these systems often encounter issues when applied to larger, more complex environments and decision making processes[2]. Thus, using large Recurrent Neural Networks (RNNs), which can be efficiently trained through backpropagation, to create a model-based RL algorithm can help improve the learner's capacity for scale and complexity[3]. However, even as large RNNs enable more complex RL models, the credit assignment problem remains a bottleneck to learning effective decision policies. Therefore, in this tutorial, we will describe a model-based agent with a highly expressive world model---to reflect the complexity of the systems in which it is intended to operate---and a very simple controller model---to minimize the size of the search space for the credit assignment problem[4].

This RL model will closely mimic the structure of human decision processes. It will consist of three distinct but interoperating components:

1. A Vision Component, which takes in 64x64 pixel frames of the environment and produces abstract, low-dimensional encodings of them;
2. A Memory Component, which, at every time step, attempts to predict the next state of the environment; and
3. A Decision-Making Component (i.e. "Controller Component"), which is a simple linear layer that maps outputs from both the Vision and Memory Components to actions that are intended to maximize the agent's rewards over time.

Each phase of the training process for this model is built upon previous phases. Later in this tutorial, we will demonstrate how to train our model on a simulated version of its environment. Training in such a simulation will allow us to control the uncertainty of the training data. By adjusting the uncertainty, we can ensure that the model is learning effectively, and we can boost its robustness and overall performance in the real world.

Architecture

The Vision Component is implemented as a convolutional variational autoencoder (ConvVAE)[5]. It receives input from the environment, and its objective is to learn an abstract, compressed representation of each input. This input is then passed through a series of convolutional layers

[1] Ha and Schmidhuber, "Recurrent World Models Facilitate Policy Evolution", 2018.
[2] http://incompleteideas.net/book/bookdraft2018mar21.pdf
[3] Ha and Schmidhuber, "Recurrent World Models Facilitate Policy Evolution", 2018.
[4] Ibid.
[5] https://www.tensorflow.org/tutorials/generative/cvae

to encode it into low-dimensional vectors μ and σ. The latent vector z is then sampled from a factored Gaussian distribution with mean μ and diagonal variance σ^2. The latent vector z is then passed through a series of deconvolutional layers to reconstruct the original image. The architecture of the Vision module is shown in Figure 1 (all convolutional and deconvolutional layers use a stride of 2).

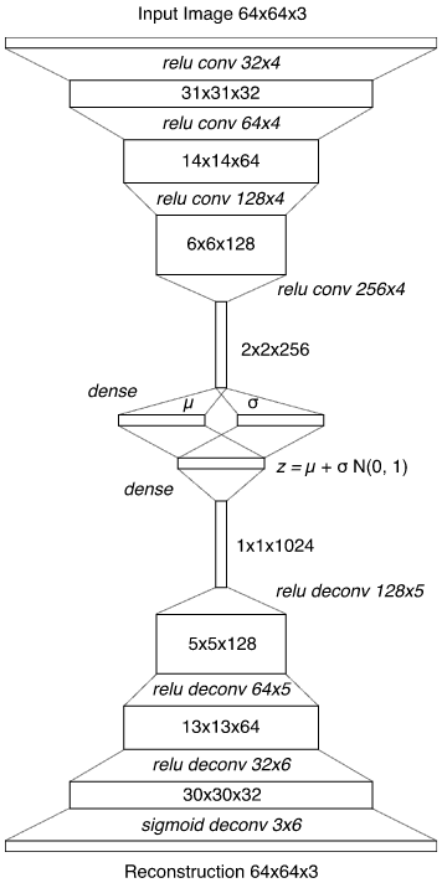

Figure 1: Vision model architecture
[Source: Ha & Schmidhuber, 2018]

The Vision Component is trained to optimize two quantities: the distance between the input image and the image reconstruction and the KL loss[6]. Training is discussed in further detail in the "Experiments" section.

Input information vectorized by the Vision Component is then passed to the Memory Component of our model. The purpose of the Memory Component is to predict future z vectors produced by the Vision Component. To that end, our Memory Component is implemented as a Long Short-Term Memory (LSTM) recurrent neural network (RNN) with a Mixture Density Network (MDN) as its output layer[7]. At each time step, the Memory Component's input consists of z_t, the output of the vision module at time t; a_t, the action of the agent at time t; and h_t, the hidden state of the LSTM RNN at time t. Using this input, the Memory Component attempts to predict z_{t+1}, the next output of the Vision Component. Many complex environments are stochastic in nature, so the MDN portion of the Memory Component outputs a probability density function $p(z_{t+1} | z_t, a_t, h_t)$, rather than a deterministic prediction[8]. The architecture of the memory module is shown in Figure 2.

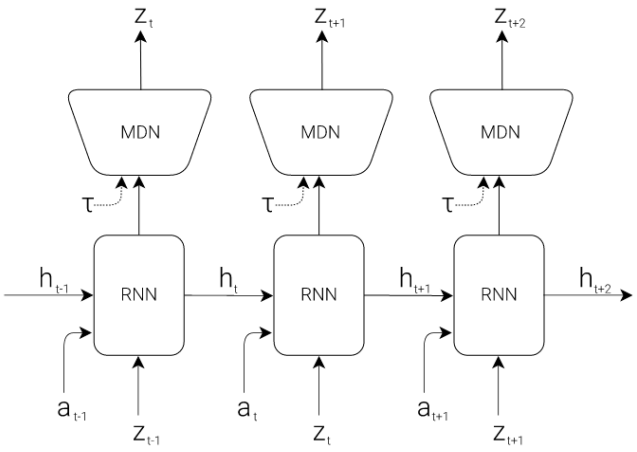

Figure 2: Memory model architecture
[Source: Ha & Schmidhuber, 2018]

Figure 2 introduces the parameter τ, which symbolizes the model's temperature parameter. The temperature parameter can be used to vary the level of uncertainty in the Memory Component's outputs, as explained in the introductory section of this tutorial. We will more explicitly address the effects of changing this parameter in the "Experiments" section.

Together, the Vision and Memory Components comprise the agent's "world model." These two components enable the agent to perceive its environment in such a way that

[6] https://wiseodd.github.io/techblog/2016/12/10/variational-autoencoder/
[7] Ha, Jongejan, and Johnson, "Draw Together with a Neural Network", 2017
[8] Ha and Schmidhuber, "Recurrent World Models Facilitate Policy Evolution", 2018.

the Controller Component---which handles decision making---can determine action policies that will maximize the agent's expected rewards over time.

The Controller Component takes in the outputs of the Vision and Memory Components and determines what actions the agent should take in order to maximize its expected cumulative rewards. In order to keep the decision process and credit assignment relatively simple, the Controller Component is implemented as a single-layer linear model that directly maps the outputs of the Vision and Memory Components to some action:

$$a_t = W_c[z_t\ h_t] + b_c$$ [9]

Equation 1: Where *W* is the weight matrix and *b* is the bias vector that we seek to optimize through training

Keeping the Controller Component simple also allows us to employ evolution strategies (ES) to optimize its weights. ES algorithms are often much more efficient than traditional RL optimization algorithms, especially in cases where rewards are sparse and credit assignment takes place over long periods of time[10]. We will use the Covariance-Matrix Adaptation Evolution Strategy (CMA-ES) as our optimization algorithm because "it is known to work well for solution spaces with up to a few thousand parameters"[11]. We will use a population size of 64, and we will define the fitness score for each agent as the average cumulative reward over 16 random rollouts of the environment.

Armed with this architecture, we can implement two experiments to test the learning capability of our model. The first experiment is a car racing experiment, where the world model will perceive a race track environment, and the Controller will control the actions of the car. The second experiment is a VizDoom experiment, where the world model will perceive the trajectories of fireballs shot by enemies in the game, and the Controller will direct the movements of the agent. We will discuss both the training process and expected performance as they relate to each experiment individually.

Experiments

Car Racing

In this experiment, racetracks will be randomly generated for each trial, and the agent will be rewarded for visiting as many tiles as possible in as little time as possible. The agent has three actions: steering left or right, accelerating, and braking. We will train each of the components of our agent separately, beginning with the Vision Component.

First, we will generate 10,000 random rollouts of the environment. For each rollout, we have an agent acting randomly to explore the environment multiple times. We record each of these

[9] Ibid.
[10] https://blog.otoro.net/2017/11/12/evolving-stable-strategies/
[11] Ha and Schmidhuber, "Recurrent World Models Facilitate Policy Evolution", 2018.

random actions and the resulting observations from the environment, and use these observations to train the Vision Component.

Next, we will use the trained Vision Component to pre-process each frame at time *t* into its latent vector, z_t. We can then use this pre-processed data, along with the actions taken by our random policy agent, to train our MDN-RNN Memory Component. In this experiment, our LSTM RNN will have 256 hidden units, and our MDN will sample its predictions from a mixture of 5 Gaussian distributions. Finally, we will use the outputs of our trained Vision and Memory Components to evolve our Controller parameters using CMA-ES, seeking to maximize the expected cumulative reward of each rollout.

As an additional step, we can use this world model to generate hypothetical racetracks that we can use to test our agent. Because our world model can predict the future, we can take its prediction at each time step, sample a single frame, and use this sample as the next input to the vision module. Thus, we can use an agent trained in a real-world environment to operate in a kind of "dream state."

VizDoom

In our second experiment, our agent must learn to avoid fireballs shot by monsters in a virtual environment. The objective is to stay alive (by staying away from fireballs) for as long as possible, so the cumulative reward will be defined as the number of time steps for which the agent remains alive in a particular rollout.

While the car racing experiment trained our agent in the "real" world and then deployed it in a dream state, the objective of the VizDoom experiment is to test whether our agent can be trained in a dream state and then successfully deployed in the "real" world. The setup for the VizDoom experiment will have some important distinctions from the Car Racing experiment. First, our Memory Component must predict both the next frame and whether our agent will die in the next frame. The latter can be predicted as a boolean event, $done_t$. In this experiment, our LSTM RNN has 512 hidden units, and our MDN again samples its predictions from a mixture of 5 Gaussian distributions. We will instantiate this environment by wrapping an OpenAI Gym environment over the Memory module. This will enable us to train our agent inside of a simulated, dream-state environment, rather than in the actual environment.

Up to this this point, the steps in the VizDoom experiment have been similar to those in the Car Racing experiment:

1. Collect 10,000 rollouts of the environment and record the actions of a random-policy agent exploring each rollout
2. Train the Vision Component to encode each frame into a latent vector
3. Train the Memory Component to model $p(z_{t+1},\ done_{t+1} |\ z_t, a_t, h_t)$
4. Use CMA-ES to optimize the Controller's parameters to maximize the agent's expected survival time inside the virtual environment

One critical difference between this experiment and the previous experiment is that we can attempt to improve the agent's learned policy by adjusting the temperature parameter. By increasing the uncertainty, we can make the dream environment more difficult to navigate. As Ha and Schmidhuber's World Models paper finds, agents that perform well in higher temperature environments tend to perform better in the real world. In addition, increasing τ helps prevent the agent from achieving artificially high scores by adopting policies that exploit imperfections in the virtual environments. For example, when τ was too low, the authors of the paper found that their agent was able to discover adversarial strategies that prevented the monsters from shooting at all. It is important to note, however, that increasing τ too much can result in an environment that is too random for the agent to learn from. The authors find that for the VizDoom dream-state environment, the optimal τ value is between 1.15 and 1.30, depending on the desired risk of the model. After learning an action policy in the dream state training environment, the agent can then be deployed in actual VizDoom scenarios.

Conclusion

In this tutorial, we described the architecture and implementation of a model-based RL agent. The agent is designed to replicate the human decision-making process through an expressive world model, which stores the memories of past experiences and predicts the future based on past and present sensory data, and a simple controller model that helps us overcome the credit assignment bottleneck without sacrificing model capacity. The RL agent is comprised of three distinct but interoperating components: a Vision Component, implemented as a ConvVAE; a Memory Component, implemented as an LSTM RNN with a Mixture Density Network as the output layer; and a Controller Component, implemented as a simple linear layer.

In the "Experiments" section, we describe two experiments implemented by Ha and Schmidhuber, and we explain the training procedures for each. These training procedures can be generalized to suit many complex reinforcement learning tasks. The generalized, iterative training procedure is detailed below:

1. Initialize the Memory and Controller Components with random parameters
2. Rollout the training environment Ntimes, and deploy the random agent. Save all actions a_t and observations x_t
3. Train the Memory Component to model $p(a_{t+1}, x_{t+1}, r_{t+1}, d_{t+1} | x_t, a_t, h_t)$ where a_{t+1} is the action taken at time $t+1$, x_{t+1} is the observation at time $t+1$, r_{t+1} is the reward received at time $t+1$, and d_{t+1} is whether or not the agent dies at time $t+1$. Not all of these predictions will be necessary for all tasks, but this probability function provides a template that can be tailored to the needs of particular problems
4. Train the Controller Component to maximize the expected cumulative rewards received by the agent.

Thus, this model-based RL paradigm can serve as the foundation for many complex machine learning tasks.

Reproducing the Results of the Car Racing Experiment

In order to reproduce the results of the Car Racing experiment, we deployed the World Models repository[12] on an AWS EC2 instance using Docker. First, we generated 2,000 random rollouts of the environment with 300 time steps each. The data generation process took approximately 30 minutes to complete. We used these rollouts to train the Vision and Memory Components of our model.

We trained the Vision Component to encode each input frame into a 32-dimensional latent vector. The Vision Component was trained to minimize the sum of the reconstruction loss and the KL loss[13]. While training the Vision Component, we encountered frequent memory issues. As a result, we reduced the number of rollouts on which we trained our model to 100 and the number of training epochs to 3. The output of the "check VAE" step is shown in Figure 3. As shown, the VAE output contains noticeably less detail than the original input, but the primary components of each frame are maintained. The training process took approximately 4 hours to complete.

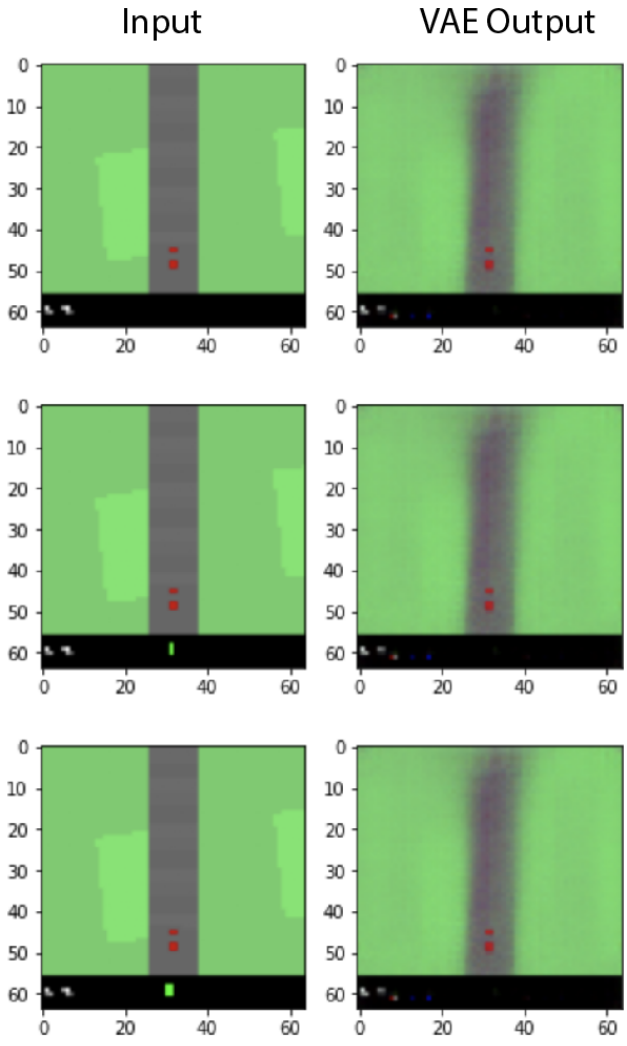

Figure 3: VAE output

Then, we used our trained Vision Component to generate data to train our Memory Component. We trained our Memory Component on 4,000 batches of the data using a batch size of 100 frames. The training process took approximately 2 hours to complete.

Finally, we used our trained Vision and Memory Components to train our Controller. We trained our controller using 4 workers with 1 trial per worker. Each generation had a population size of 4 * 1 = 4. In each generation, the agent's fitness score was defined as its average reward over 16 random rollouts of the environment. We trained our model for approximately 2,500 generations with a run-time of 2.5 days. We aimed to match the performance of Ha and Schmidhuber, who trained their model for approximately 2,000 generations using a population size of 64. The rolling average scores of our agents, computed with a window size of 100 generations, are plotted in Figure 4.

[12] https://github.com/pantelis-classes/world-models-latest
[13] https://wiseodd.github.io/techblog/2016/12/10/variational-autoencoder/

In our final generation, we achieved a maximum fitness score of approximately 111. While we did not match the performance of Ha and Schmidhuber, we did achieve significant improvement over the course of our training. However, our evolutionary path appears much less stable than that shown in the World Models paper. We believe the variability in our graph is primarily due to the small population size of our generations, which likely made it difficult for our CMA-ES optimizer to evolve towards an optimal policy in smooth fashion. With a larger population size or additional episodes per agent, our policy may have evolved more quickly towards an ideal strategy.

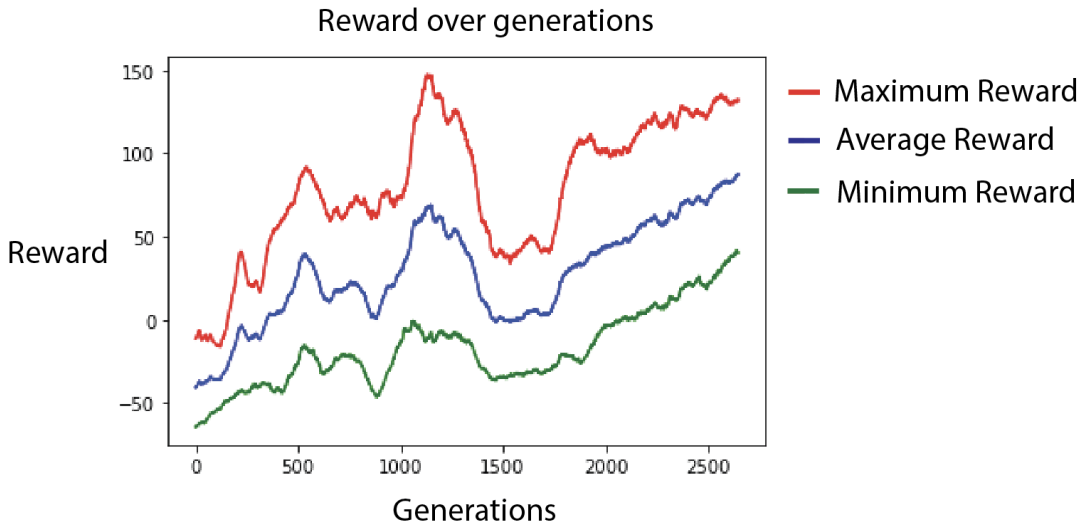

Figure 4: Reward over generations

A video of our CarRacing model fully deployed is available in our Github repository[14]. As shown in the video, the agent is able to correctly avoid driving onto the grass as it travels along a straight path; however, the agent has difficulty navigating sharp turns, and once it leaves the track, it cannot efficiently return to it. This deficiency may be attributable to a combination of the limited population size of the CMA-ES generations, which hindered our policy optimization, and undertraining the RNN.

[14] https://github.com/haydenedelson/world-models-latest/tree/master/artifacts

Documenting a new approach using a VAE/GAN

In an attempt to improve the capacity of the agent's world model, we augmented the Vision Component's ConvVAE, with a generative adversarial network (GAN)[15]. In this augmented approach, the encoder of the VAE maintains its original purpose---to learn a compressed, abstract representation of each input frame---but the decoder is replaced by the generator of a GAN, and the discriminator is used to train the encoding and decoding processes. A general depiction of this architecture is shown in Figure 5.

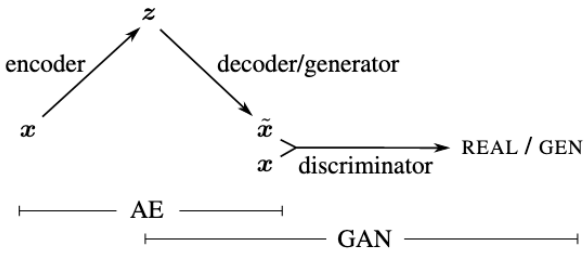

Figure 5: VAE/GAN architecture [Source: Lindbo et al., 2016]

In our approach, we integrate the architectures of the original Ha and Schmidhuber vision component with the VAE/GAN architecture proposed in Larsen et al[16]. Our model architecture is delineated in Figure 6. The primary design decisions are documented below:

1. We use the encoder architecture described in Larsen et al. However, in keeping with the Vision Component's stated purpose of learning a low-dimensional representation of each input frame, we maintain the low dimensional size and stochastic nature of the encoder used in Ha and Schmidhuber's World Models paper.
2. Our generator architecture incorporates elements from both papers, but it is closer overall to the decoder architecture described in Ha & Schmidhuber. This architecture more efficiently recreates an image of the desired output size.
3. Our discriminator architecture is based on the discriminator architecture described in Larsen et al.
4. In our training loop, we apply the loss functions defined in Larsen et al., which were derived specifically for VAE/GAN training

We trained our Vision model for five epochs on 300 rollouts of the environment. Unfortunately, this was not enough time for the model to achieve optimal performance. However, we were able to document notable progress based on the model's loss metrics, and we believe that with

Encoder	Generator	Discriminator
Input (64x64x3)	Input (32x?)	Input (64x64x3)
5x5 64 Conv2D, BN, ReLU	1024 Dense, BN, ReLU	5x5 32 Conv2D, ReLU
5x5 128 Conv2D, BN, ReLU	Reshape (1x1x1024)	5x5 128 Conv2D, BN, ReLU
5x5 256 Conv2D, BN, ReLU	5x5 62 Conv2D Transpose, BN, ReLU	5x5 256 Conv2D, BN, ReLU
Flatten	5x5 64 Conv2D Transpose, BN, ReLU	5x5 256 Conv2D, BN, ReLU
1024 Dense, BN, ReLU	6x6 32 Conv2DTranspose, BN, ReLU	Flatten
32 Dense: mu	6x6 3 Conv2DTranspose, tanh	256 Dense, BN, ReLU
32 Dense: log var		1 Dense, sigmoid
Sampling: z		

Figure 6: Our implemented VAE/GAN model architecture

[15] https://www.tensorflow.org/tutorials/generative/dcgan
[16] Larsen, Sonderby, Larochelle, and Winther, "Autoencoding beyond pixels using a learned similarity metric", 2016

additional training time, the model could perform very effectively. The average KL loss decreased from 0.164 in the first epoch to 0.136 in the fifth epoch. The average encoder loss decreased from 0.021 to 0.016, and the average generator loss improved from -0.801 to -0.729 over the same time period. The average discriminator loss did not change significantly.

However, we believe that with additional training time, discriminator loss would have improved. In the final epoch, we increased the learning rate from 0.0001 to 0.0003, and this change appeared to be highly beneficial: the generator loss began to improve much more rapidly, and the discriminator loss, which had gone unchanged for several epochs, began once again to decrease slightly. We believe that these trends could have continued if given additional time.

Additionally, with more time, we would have experimented with the following strategies to improve the performance of our VAE/GAN: firstly, we would have increased the size of our dataset by importing additional rollouts. Several GAN papers have shown that as the size or complexity of the GAN architecture increases, the size of the dataset should increase as well[17] [18]. The authors of the World Models paper trained their VAE on 2000 rollouts of data, but due to time and compute limitations, we truncated our training set to only 300 rollouts. It is possible that the reason the discriminator loss stopped decreasing was that the GAN fell into some form of mode collapse and stopped learning effectively. A larger and more diverse dataset could have helped our model avoid this failure. Moreover, as was noted previously, increasing the learning rate from 0.0001 to 0.0003 resulted in some improvements. Larsen et al. employ a learning rate of 0.0003, so with more time, we would have closely observed the impacts of the higher learning rate. And finally, we could have tried RMSProp optimizers for our models, rather than Adam optimizers. Larsen et al. and the authors of the “Wasserstein GAN” paper[19] train their models with RMSProp optimizers, so perhaps RMSProp would have converged more efficiently.

Link to Github repository: https://github.com/haydenedelson/world-models-latest

[17] Liu and Hu, “GAN-Based Image Data Augmentation”, 2019
[18] Arjovksy, Chintala, and Bottou, “Wasserstein GAN”, 2017
[19] Ibid.